V.09.

Общий обзор занятия.

В прошлый раз мы прикоснулись к рекурсии. Прикоснулись, видимо, наиболее естественным для ней образом - через деревья. Деревья самые простые, ну, или почти самые простые - бинарные деревья, хотя и позволили себе некоторые вариации (имея в виду дерево поиска, да ещё со счётчиками размера поддеревьев). Сегодня мы влезаем в рекурсию по возможности поглубже. Именно поглубже, основной предмет сегодня - рекурсия, и хочется, чтобы дети её даже не столько поняли, понять - требуется время, сколько воспринять рекурсию как можно более глубже в себя. Не заморачиваемся сегодня особо сложными задачами в техническом, реализационном, идейном смысле. А целимся строго в рекурсию. И развиваем идеи рекурсии во всех направлениях, намеченных в прошлый раз:

- сведение задачи к такой же задаче либо разбиение на несколько таких же задач, но на меньшем объёме данных
- рекурсивное определение объекта прямо связано, естественным образом, с рекурсивной обработкой объекта
- линия деревьев, которые по сути своей рекурсивны

Добиваемся этого, рассматривая побольше мелких, несложных технически примеров, в которых рекурсия естественна, по крайней мере должна выглядеть естественной, показывать эти примеры, рассматривать их именно “влёгкую”, стараясь проскользнуть, где это возможно, акцентируясь на рекурсии.

Хочется воссоздать вот то «лёгкое» настроение, которое мы старались создать в прошлый раз. Т.е. рекурсия – это легко, просто, ясно, только не надо ей внутрь заглядывать, уводить детей от этого (а они будут всячески пытаться туда залезть - нерекурсивные привычки в них сильны, они ведь привыкли трассировать код, явно или неявно), смотреть на рекурсию надо только «сверху» - **функция делает вот то-то `с некоторой_частью/ подмножеством/ меньшим_объектом`, значит эта функция делает то же самое с `целым/ всем_множеством/бóльшим_объектом`, ну, или наоборот: чтобы сделать что-то с `бОльшим_объектом` надо сделать то же самое плюс, возможно, ещё что-то с**

`меньшим объектом или несколькими меньшими объектами`). **Причём процесс обязательно закончится, потому что он**

> **1. каждый раз вызывает меньший (точнее - более близкий к терминальному) случай**
> **2. имеются терминальные случаи, к которым мы обязательно придём рано или поздно; терминальный случай - это тот, который обрабатывается совсем просто, настолько просто, что его можно написать, не особо задумываясь, просто втупую взять и обработать.**

Момент обязательного окончания рекурсивного вызова надо постоянно озвучивать, произносить явно, подчёркнуто так, и неоднократно. Что вот, мы сводим задачу к задаче с меньшим размером объекта, а задача с меньшим объёмом данных автоматически сводится к задаче (или задачам) с ещё меньшим, а та (те) - к ещё меньшим и т.д. Но при этом где-то этому должен наступить конец. И наступает он тогда, когда появляется задача с настолько малым размером данных, что мы можем их обработать явно, при этом более или менее легко и просто. А это и есть терминальный случай.

Лекция

Громкая песня звучит всё занятие: рекурсия − как это хорошо! И исполняем мы её на различных и разнообразных примерах. Все примеры естественным образом вписываются в вышеописанную схему. Не буду её цитировать - она приведена двумя абзацами выше и выделена болдом.

Даже не знаю, с какого примера начать, глаза разбегаются - всё такое вкусное...

Угадайка

Начнём, пожалуй с совсем простого примера: любимая игра всех времён и народов - “Угадайка”. Правила простые и всем известные: один игрок задумывает натуральное число от 1 до 100. Другой угадывает его, называя разные числа. Первый игрок отвечает “задуманное число меньше”, “задуманное число больше”

либо “угадал”. В нашем примере пользователь - это первый игрок, а за второго игрока играет программа - она угадывает.

Совершенно естественная рекурсия, лежащая прямо на поверхности: объект обработки - отрезок, на котором может находиться неизвестное число (которое программа старается угадать). В результате проверки отрезок уменьшается (этот факт надо тщательно проверить - убедиться, что отрезок действительно сокращается), приближаясь тем самым к терминальному случаю. Любой шанс пройти вызов функции guess без сокращения отрезка выкинет нас в бесконеную рекурсию, тот самый порочный круг, вечное движение типа “стол - это стол”. А терминальный случай здесь - успешная попытка. Понятно, что она может случиться в любой момент, но когда подозрительный отрезок сократится до одного числа, угадывание неизбежно, а отрезок каждый раз сокращается. Так что терминальный случай случится обязательно.

Вот код: **пример** 01.guess.go.

```
package main

import "fmt"

func guess(left, right int)  {
    var answer string
    c:= (left + right)/2
    fmt.Println("my shot: ", c)
    fmt.Scanln(&answer)
    switch answer {
    case "<":
        guess(left, c-1)
    case ">":
        guess(c+1, right)
    case "=":
        fmt.Println("Congratulations!")
        return
    default:
        fmt.Println("Incorrect answer")
        fmt.Println("Enter \"<\" if your secret number <", c)
```

```
        fmt.Println("Enter \">\" if your secret number >", c)
        fmt.Println("Enter \"=\" if your secret number is", c)
        guess(left, right)
    }
}

func main()  {
    guess(1, 100)
}
```

Заметим, что эта программа подразумевает совершенную добросовестность пользователя.
Хорошее маленькое задание: переделать программу так, чтобы она отлавливала недобросовестного пользователя - игрока, который может говорить неправду.
Подсказка. В таком случае возможно сужение подозрительного отрезка до нуля: `left>right`.
Коварный вопрос: а чем это лучше программы с циклом вместо рекурсии? А ничем, честно говоря, ну, почти ничем. Более того, в определённом смысле, хуже, но об этом пока не говорим, даже не смотрим в ту сторону. Об этом поговорим через пару-тройку занятий, когда пойдёт разговор об опасностях рекурсии.
Ещё одно задание: написать функцию guess с циклом вместо рекурсии. Кроме того, обеим функциям давать доступ к угадываемому числу, но чтобы функция честно его угадывала. При этом убрать из функций весь интерфейс пользователя - пусть функция сама сравнивает свою (честно вычисленную попытку) с угадываемым числом и ничего не возвращает, просто заканчивается при угадывании. Разумеется, реакцию на некорректный ответ тоже можно (и нужно) убрать. Зачем так делать? А чтобы сравнить скорость угадывания, скорость работы этих функций. Бенчмарк, конечно. Так что написать эти функции и побенчмарчить их.

На всякий случай приведу код этих функций, пусть будет, но давать его детям сразу не стОит.
Код: **пример 01.guessA.go.**

```
package main
```

```
var secret int = 73

func guess(left, right int) {
    // recursive variant
    attempt:= (left + right) / 2
    switch {
    case secret < attempt:
        guess(left, attempt-1)
    case secret > attempt:
        guess(attempt+1, right)
    case secret == attempt:
        return
    }
}

func guess2(left, right int) {
    // cyclic variant
    for {
        attempt:= (left + right) / 2
        switch {
        case secret < attempt:
            right = attempt - 1
        case secret > attempt:
            left = attempt + 1
        case secret == attempt:
            return
        }
    }
}

func main() {
    guess(1, 100)
    guess2(1, 100)
}
```

Наибольший общий делитель – алгоритм Евклида.

Напомним вкратце сам алгоритм. gcd - greatest common divisor - это НОД, наибольший общий делитель.

Основное соотношение. N и K - натуральные числа, причём

```
    Если N > K, то  gcd(N, K) = gcd(N-K, K).
```

Отсюда сразу следует алгоритм Евклида с вычитанием.

```
Вход: имеется два натуральных (целых положительных) числа: N и K.
- Если N = K, то gcd(N, K) = N, и алгоритм заканчивается.
- Если N > K, то   N = N - K, иначе   K = K - N
- Перейти на начало.
```

У этого варианта есть один существенный недостаток: если вдруг N - очень большое число, а K - маленькое, то мы будем очень много раз уменьшать N на K, и алгоритм будет работать очень медленно. А что происходит, когда N много больше K? Мы вычитаем K, вычитем, вычитаем до тех пор, пока от N не останется величина, меньшая K. Понятно, что в итоге от N останется остаток от деления N на K. Ну, так и давайте сразу так и сделаем. И заодно, поскольку после этого N станет меньше K, то сразу поменяем их местами. В итоге получаем следующий вариант:

Алгоритм Евклида с делением

```
Вход: имеется два натуральных числа: N и K.
- Если K = 0, то gcd(N , K) = N, и алгоритм заканчивается.
- N, K = K, N % K
- Перейти на начало
```

Конечно, надо обратить внимание на то, что процесс оканчивается при K = 0, а не при N = K. И всё хорошо, кроме одной мелкой зацепки: а если в начале N<K? Но ничего страшного на происходит: первый проход просто поменяет N и K местами.

И рекурсия здесь совершенно аналогичная рекурсии в “Угадайке” - каждый раз одно из чисел уменьшается, ну, или чтобы проще - сумма чисел на входе уменьшается. При этом у нас никогда не получаются отрицательные числа - просто по алгоритму (в первом варианте даже и нулей не получается, а во втором

- ноль появляется только в самом конце). Следовательно процесс не может продолжаться бесконечно, т.е. он таки действительно обязательно придёт к терминальному случаю. Интересный вариант - не потому заканчивается, что обязательно придёт в терминальному случаю, а потому придёт к терминалу, что не может длится бесконечно.
Вот код: **пример** 02.euclid.go.

```
package main

import "fmt"

func gcd1(a, b uint64) uint64  {
    if a == b  { return a }
    if a > b  { return gcd1(a-b, b) }
    /*if a > b */ return gcd1(a, b-a)
}

func gcd2(a, b uint64) uint64  {
    if b==0  {return a}
    return gcd2(b, a%b)
}

func main()  {
    var a, b uint64
    fmt.Print("Enter first number:  ")
    fmt.Scan(&a)
    fmt.Print("Enter second number:  ")
    fmt.Scan(&b)
    fmt.Println ( gcd1(a, b) )
    fmt.Println ( gcd2(a, b) )
}
```

И ещё один простой пример рекурсии -

Обработка связного списка

А вот здесь всё получается очень похоже на то, что было с бинарными деревьями, только проще. Вводим **рекурсивное определение связного списка**: Связный список - это

```
- пустое множество  ИЛИ
- вершина, к которой прилеплен связный список
```

Выглядит вполне нормально на фоне прошлого занятия. А рекурсивное определение естественным образом перетекает в рекурсивную реализацию обработки этой структуры.
Определим типы естественным образом:

```
type (
    list struct {
        head *lmnt
    }
    lmnt struct {
        x int
        next *lmnt
    }
)
```

и набросаем для примера совсем простую функцию, возвращающую размер списка. Следуя за определением, получаем: длина пустого списка равна 0, длина непустого списка равна 1 плюс длина списка, следующего за начальным элементом. Вот код:

```
func (s list) Len() int  {
    if s.Empty()  {
        return 0
    }  else  {
        return 1 + list{(*s.head).next}.Len()
    }
}
```

Всё! Сорри, не удержался от нелюбимого восклицательного знака.

Аалогично получаем идеи ещё нескольких методов.

1. Добавить элемент в конец списка. Если список пустой, то делаем одноэлементный список. Если список непустой, то берём список, пристроенный к начальному элементу, добавляем элемент в конец этого списка, пристраиваем полученный список к начальному элементу исходного списка.
2. Вывести список в порядке следования. Если список пустой, то ничего не выводим, но давайте всё-таки отпечатаем пустую строку с переводом курсора на следующую строку. Если непустой, то выводим начальный элемент, а затем весь оставшийся список. Тонкий момент - в конце курсор перейдёт на следующую строку.
3. Вывести список в обратном порядке. Если список пустой, то ничего не делаем. Если непустой, то выводим в обратном порядке весь список, следующий за начальным элементом, а затем выводим начальный элемент. А как здесь перевести курсор в конце? Ну, можно просто сделать это вне рекурсивной функции.

В общем, полный код вот: **пример 03.list.go.**

```
package main

import     (
    "fmt"
)

type  (
    list struct {
        head *lmnt
    }
    lmnt struct {
        x int
        next *lmnt
    }
)

func initList() list  {
    return list{nil}
}
```

```
func (s list) Empty() bool  {
    return s.head == nil
}

func (s list) Len() int  {
    if s.Empty()  {
        return 0
    }  else  {
        return 1 + list{(*s.head).next}.Len()
    }
}

func (s list) PrintThere()  {
    if s.Empty()  {
        fmt.Println()
    }  else  {
        fmt.Print((*s.head).x, " ")
        list{(*s.head).next}.PrintThere()
    }
}

func (s list) PrintBack()  {
    if !s.Empty()  {
        list{(*s.head).next}.PrintBack()
        fmt.Print((*s.head).x, " ")
    }
}

func (s *list) Add(n int)  {
    (*s).head = &lmnt{x: n, next: (*s).head}
}

func (s list) AddBack(n int) list {
    if s.Empty() {
        return list{head: &lmnt{x: n, next: nil}}
```

```
    } else {
        tail:= list{(*s.head).next}.AddBack(n)
        (*s.head).next = tail.head
        return s
    }
}

func main() {
    l:= initList()
    for _, x := range []int{2, 5, 4, 1, 2, 8} {
        l.Add(x)
    }
    fmt.Println(l.Len())    // 6
    l.PrintThere()          // 8 2 1 4 5 2
    l.PrintBack()           // 2 5 4 1 2 8
    fmt.Println()
    l = initList()
    for _, x := range []int{2, 5, 4, 1, 2, 8} {
        l = l.AddBack(x)
    }
    fmt.Println(l.Len())    // 6
    l.PrintThere()          // 2 5 4 1 2 8
    l.PrintBack()           // 8 2 1 4 5 2
    fmt.Println()
}
```

Очень много ещё упражнений на списки можно реализовать вот таким образом. Вообще-то, разнообразные упражнения на списки приведены в 5-ом занятии. Многие из них довольно естественно реализуются рекурсивно. Список таких заданий я приведу здесь в заданиях на практику.

И, напоследок, вечный великий и совершенно рекурсивный сюжет:

Обход дерева каталогов.

Как обойти каталог со всем его содержимым? Берём его и читаем. Это как? А очень просто, ведь каталог - это список всех файлов и подкаталогов данного

каталога. Внимание - рекурсия: каталог это список файлов и подкаталогов. И всё прекрасно получается. Читаем эти записи (как именно - это неважно) и просматриваем их последовательно. Если запись соответствует файлу, то обрабатываем соответствующий файл, например, выводим его название, если же запись соответствует каталогу, то надо обойти этот каталог. А как это сделать? Да именно так, как мы только что описали. Всё! Рекурсия заходит на этом примере предельно естественно и мягко.

Ну, и дальше остаются в основном технические вопросы по этому сюжету. Читать каталог можно по-разному, я привожу пример чтения каталога с помошью функции `ioutil.ReadDir`:

```
func ReadDir(dirname string) ([]os.FileInfo, error)
```

ReadDir reads the directory named by dirname and returns a list of directory entries sorted by filename.
Конечно же, надо напомнить про тип os.FileInfo - давно это было:

```
type FileInfo
```

содержит некоторую информацию о файле, в частности:

```
    Name() string       // имя файла
    Size() int64        // длина в байтах для обычных файлов; в остальных случая
х зависит от конкретной ОС
    ModTime() time.Time // время последнего изменения файла
    IsDir() bool
```

В результате получаем простой, ясный и естественный код: **пример 04.dirA.go.**

```
package main

import (
    "fmt"
    "io/ioutil"
    "os"
)
```

```
func main() {
// Выводит список всех файлов и каталогов в каталоге,
// переданном в командной строке, и во всех его подкаталогах.
// По умолчанию - начиная с текущего каталога.
    var headDir string
    if len(os.Args) == 1  {
        headDir = "."
    }  else  {
        headDir = os.Args[1]
    }
    traceDir(headDir, "")
}

func traceDir (startDir string, prefixStr string)  {
    if files, err := ioutil.ReadDir(startDir); err == nil {
        for _, f := range files {
            fmt.Println(prefixStr, f.Name())
            if f.IsDir()  {
                traceDir(startDir+"\\"+f.Name(), prefixStr+"  ")
                // Внимание! "\\" - это строка из одного символа \
            }
        }
    }
}
```

Запустим, посмотрим, всё хорошо. Но не совсем. Нехорошо то, что содержимое каталога выводится вразнобой: выводим названия каких-то файлов, потом отвлекаемся на подкаталог (со всеми его подкаталогами, а это может быть долго), потом снова выводим названия файлов из текущего каталога, снова отвлекаемся. снова возвращаемся - неудобненько получается. А что делать, чтобы вывести сначала названия всех файлов текущего каталога, а потом заныривать в подкаталоги? Ну, так ответ уже есть, именно это и делать: сначала выводить названия всех файлов текущего каталога, а потом заныривать в подкаталоги. Да, для этого понадобится проходить по каталогу два раза, но это и всё. Код изменяется несущественно: **пример** 04.dirB.go.

```
package main

import (
    "fmt"
    "io/ioutil"
    "os"
)

func main() {
// Выводит список всех файлов и каталогов в каталоге,
// переданном в командной строке, и во всех его подкаталогах.
// По умолчанию - начиная с текущего каталога.
    var headDir string
    if len(os.Args) == 1  {
        headDir = "."
    }  else  {
        headDir = os.Args[1]
    }
    traceDir(headDir, "")
}

func traceDir (startDir string, prefixStr string)  {
    if files, err := ioutil.ReadDir(startDir); err == nil {
        for _, f := range files {
            if !f.IsDir()  {
                fmt.Println(prefixStr, startDir + "\\" + f.Name())
            }
        }
        fmt.Println()
    }
    if files, err := ioutil.ReadDir(startDir); err == nil {
        for _, f := range files {
            if f.IsDir()  {
                traceDir(startDir+"\\"+f.Name(), prefixStr)
            }
        }
```

```
    }
}
```

Хватит на сегодня. Хотя... Есть же ещё задания на практику.

Практика.

- Хорошее маленькое задание - переделать “Угадайку” так, чтобы она отлавливала недобросовестного пользователя - игрока, который может говорить неправду - уже обсуждалось выше, так что повторять не стану. Но задание есть.
- Простая (по крайней мере в техническом аспекте) задача.

Множество натуральных чисел A = 1, 3, 4, 7, 9, 10, 13, 15, 19. 21, 22, ... определяется так:

```
- 1 входит в A;
- если x входит в A, то в A входят (2x+1) и (3x+1)
```

Написать булевскую функцию, которая определяет, входит ли число в множество A.

```
func IsA (n int) bool
```

Обсуждение задачи. Есть число n. При каких обстоятельствах оно входит в множество A? Во-первых, оно входит в A, если n = 1. Во-вторых, если у числа n имеется предшественник, который его приводит в A. Обозначим этого предшественника x. Тогда выполняется условие 2x+1=n или условие 3x+1=n, либо оба сразу. Что значит первое условие? Оно означает, что (n-1) делится на 2, и x=(n-1)/2 входит в A. Аналогично, второе условие означает, что (n-1) делится на 3, и x=(n-1)/3 входит в A. Если хотя бы одно из этих условий выполняется, то IsA(n) возвращает true (иначе - false, разумеется). Всё, если написать ещё подробнее, то задание просто умрёт.

- И, наконец, просто куча заданий на связные списки:

Некоторые упражнения на связные списки.

01. Максимум списка.

```
func (l list) Max () int
```

02. Минимум списка.

```
func (l list) Min () int
```

03. Совпадает ли список l2 с ресивером l.

```
func (l list) CompareList ( l2 list) bool
```

04. Возвращает k-й элемент списка, k >= 0. Как и в предыдущей функции, считаем, что элементы в списке пронумерованы с головы списка, и нумерация начинается с 0. Можно предусмотреть, что функция будет возвращать ошибку, если k слишком большое - больше длины списка. Либо возвращать bool - удалось или не удалось вставить.

```
func (l list) GetKth (k int) data
```

05. Создает в куче копию списка и возвращает его.

```
func (l list) CopyList list
```

06. Прицепляет список list2 к концу списка l и возвращает результат.

```
func (l list) ConcatList (list2 list) list
```

07. Вернуть номер первого элемента списка, равного x. Если такого элемента в списке нет, то вернуть -1.

```
func (l list) First (x data) int
```

08. Возвращает список, из которого исключены все элементы, равные x.

```
func (l list) DeleteX (x data) list
```

09. Возвращает список, развёрнутый задом наперёд.

```
func  (l list) Revers () list
```

10. Количество вхождений числа x в список.

```
func (l list) CountX (x int) int
```

11. Возвращает список, в котором все элементы, равные x, заменены y.

```
func (l list) SubstX (x int, y int) list
```

12. Возвращает список, полученный следующим образом: если в исходном списке подряд идут несколько одинаковых элементов, то убирает все повторы, оставляя от них только один элемент. Например, список 2 5 5 3 2 5 2 2 2 2 становится списком 2 5 3 2 5 2.

```
func (l *list) PurgeRepetitions1 () list
```

13. Список l упорядочен в неубывающем порядке. Вставляет x в список так, чтобы порядок не нарушился, и возвращает результат:

```
func (l list) InsertInOrder1(x int) list
```

14. Исходный список l упорядочен по возрастанию и не содержит повторений. Если числа x нет в списке, то вставить его в список, не нарушая порядка; если же число

```
x есть в списке, то не изменять список. Вернуть результат.
   func (l *list) InsertInOrder1(x int)
```